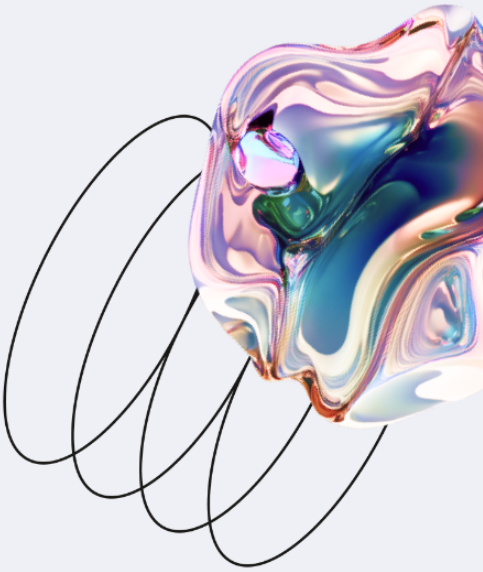
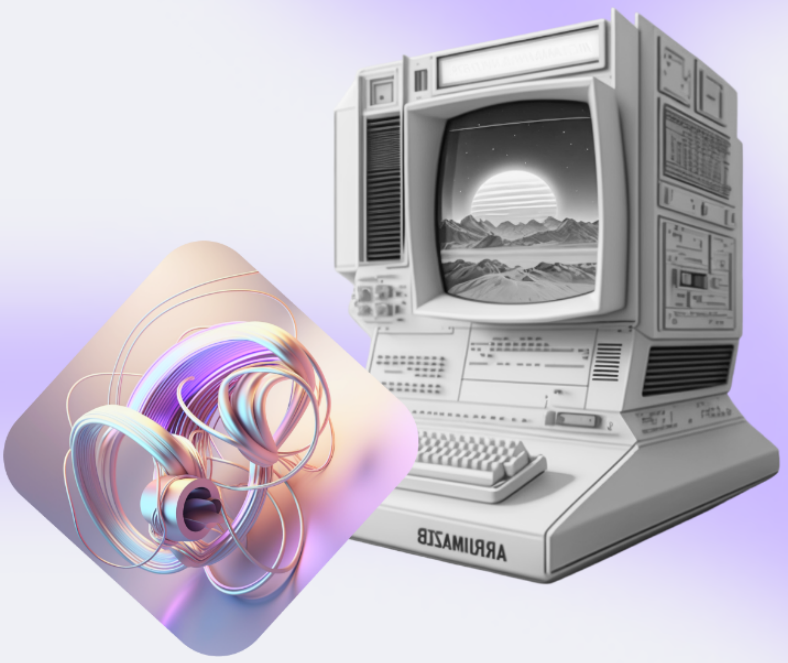
GeekBrains



Коллекции и итераторы. Модули

Продвинутый JavaScript



Оглавление

Введение

Приветствую, уважаемые студенты!

Мы начинаем курс продвинутого JavaScript, состоящего из трёх занятий. На этих занятиях мы рассмотрим темы работы коллекций, итераторов, модулей, промисов, генераторов. Всё это мы будем рассматривать с учётом самых современных стандартов языка ECMAScript. Рассмотрим также новые операторы в языке и продвинутую работу с функциями и классами с учётом самых последних новшеств.

На этом уроке мы познакомимся со специальным фоновым процессом — сборщиком мусора, который помогает освободить память от ненужных данных. Затем мы разберём продвинутую работу с итераторами и познакомимся с коллекциями Map и Set и их «собратьями» WeakMap и WeakSet.

В заключении занятия мы познакомимся с модулями JavaScript, научимся подключать модули и экспортировать из них данные.

Сборка мусора

При выполнении скрипта движком JS расходуется большое количество памяти. Вызываются функции, выполняются циклы, пересобираются объекты, на все эти операции интерпретатор должен выделять целые области памяти для хранения данных.

А что происходит, когда какая-то переменная становится не нужна? Функция выполнила свою работу, объявленные переменные внутри замыкания более недоступны ни для одной операции и хранить их в памяти не имеет никакого смысла. И если движок JS не освободит память, она в скором времени может закончиться, и это приведёт к краху нашей программы.

Главный принцип управлением памятью в JS — это её достижимость. Есть множество достижимых значений по умолчанию. Эти значения называются корневыми. Они не могут быть удалены. Среди них:

- Глобальные переменные
- Текущая функция, которая находится на стадии выполнения
- Другие функции в текущей цепочке вызовов, их параметры и локальные переменные

Далее вычисляются иные достижимые значения. Вычисляются они по ссылкам от корневых, потом по ссылкам от достижимых и так далее. В итоге, список достижимых значений напоминает ветвистое дерево, в котором от корневых значений по ссылкам можно пройти к любому нужному нам значению.

В JavaScript, как и в любом другом языке программирования, есть фоновый процесс, который управляет удалением ненужных объектов в памяти. Он называется сборщиком мусора (garbage collector). Он отслеживает все объекты, недоступные по ссылкам и удаляет их.

На предыдущих уроках мы изучали, что объекты у нас записываются в переменные по ссылке.

Например, если в объекте:

```
let obj = {
    animal: 'dog'
}
```

Переменную obj перезаписать другим значением:

```
obj = null;
```

Ссылок на наш объект более не останется, и он станет недостижим. В итоге, он падёт жертвой сборщика мусора, который освободит занимаемую им память.

Если мы создадим несколько ссылок на объект, записав их в разные переменные, то объект удалится только при удалении всех ссылок на объект.

Может возникнуть такая ситуация, когда несколько объектов имеют ссылки друг на друга, но ссылки на корневые объекты удалены. Тогда все эти объекты тоже удаляются.

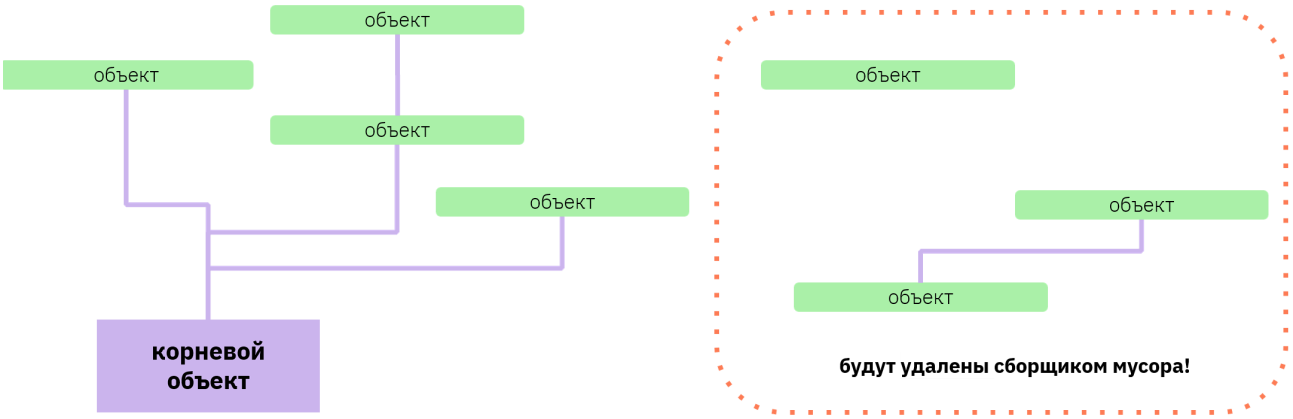
Алгоритмы работы сборщика мусора

Алгоритм работы сборщика мусора работает по принципу пометок. По этому алгоритму сначала помечаются корневые объекты, далее алгоритм идёт по ссылкам из них, далее — по ссылкам от этих ссылок.

Какой из ранее изученных методов работы он нам напоминает?

Конечно же, это рекурсия. Чтобы перебрать все объекты и не забыть ничего, нужно запомнить всё, что ты проходил и перейти по всем ссылкам, а далее — по ссылкам ссылок до самого конца.

В языке C есть специальный тип данных — ссылки, на основе которого можно строить подобные структуры данных и рассматриваются способы их обхода, которые напоминают работу нашего сборщика мусора. Там подобные структуры данных называются деревьями. Действительно, такая структура напоминает деревья, корневые объекты — ствол у основания корня. И пока данные достижимы, они «растут» на нашем дереве.



В итоге, у нас останутся какие-то объекты, которые этот алгоритм не пометил, так как к ним нет доступа по ссылкам от корней. Они будут удалены.

Естественно, такая работа при больших объёмах данных может существенно влиять на производительность выполнения нашей программы. Поэтому есть некоторые оптимизации работы сборщика мусора:

Работа сборщика мусора по возможности производится во время простоев работы скрипта.

Когда объектов много, сборщик мусора старается их разделять на части. Как в аналогии с деревом, сборщик разделяет крупные ветви дерева и проходит их в разное время. В итоге, у нас вместо одного большого обхода получается множество маленьких обходов. И вместо одной большой потери производительности множество маленьких.

Объекты разделяются по поколениям — на новые и на старые. Зачастую объекты живут недолго — функция вызвалась, создала объекты, обработала их, вернула какое-то значение и завершилась. Поэтому есть смысл проверять чаще только новые объекты. Старые проверяются реже.

Есть и иные оптимизации работы движков, которые нет смысла рассматривать в нашем курсе. Более того, движки совершенствуются, на смену одним оптимизациям приходят другие и за всем не уследить.

Symbol. Подробно

В курсе JS про ECMAScript говорилось, что в последних спецификациях EcmaScript был введён новый тип данных Symbol выполняющий роль уникального идентификатора. Ранее в ключах свойств объекта можно было использовать только строки, с введением типа Symbol появилась возможность использовать и его. Многие библиотеки, в частности, библиотека управлениями состояний Redux использует его в полной мере у себя под капотом. Разберёмся, что этот тип данных из себя представляет.

Символ объявляется через функцию Symbol():

```
const symbol = Symbol();
```

При желании можем дать описание идентификатору, которое мы сможем впоследствии использовать для того, чтобы понимать, для какого ключа создавался этот идентификатор:

```
const dogID = Symbol('dog');
```

При каждом создании символа, его значение уникально, даже если мы создадим несколько символов с одинаковым описанием:

```
const dog1 = Symbol('dog');
const dog2 = Symbol('dog');
```

Следующее выражение нам выдаст в консоли false:

```
console.log(dog1 == dog2);
```

Содержимое переменной, в которой записан символ, посмотреть нельзя. Это сделано для защиты на уровне языка, чтобы подчеркнуть разницу между символами и строками. Неявное преобразование типов также не сработает:

```
1 const dogID = Symbol('dog');
2 alert(dogID) // TypeError: Cannot convert a Symbol value to a string
```

Console.log не конвертирует тип в строку, поэтому он выведет значение переменной dogID:

```
1 console.log(dogID); // Symbol(dog)
```

Символы, в отличие от других типов данных, нельзя неявно преобразовать в другие типы. Есть у символов метод toString(), но он преобразует в строку вида, что нам дал и console.log().

Если мы хотим посмотреть только описание, можно обратиться к свойству description:

```
1 console.log(dogID.description); // dog
```

При создании объекта, где в качестве ключа у нас использован символ, необходимо использовать квадратные скобки. То же самое, при обращении к полю объекта, в названии которого использован символ. Это объясняется тем, что нам в качестве ключа нужно использовать значение переменной id, а не строку «id»:

```
1 let id = Symbol('dogID');
2 let buddy = {
3     [id]: 'Жучка'
4 }
5 console.log(buddy[id]); //Жучка
```

Создание «скрытых» свойств объектов

С использованием идентификатора, созданного с помощью Symbol, можно создать так называемые «скрытые» свойства объекта. Эти свойства не получится нечаянно перезаписать, обратившись из различных частей программы.

Разберём, чем же лучше использовать Symbol(‘id’), а не строку ‘id’? Если объект buddy будет частью стороннего кода, то небезопасно добавлять к нему новые поля.

Итераторы, работающие с этим кодом, могут его обработать и там могут возникнуть ошибки. Если мы будем использовать символы, то к ним сложнее обратиться нечаянно. Сторонний код их вряд ли увидит и, тем более, обработает, что поможет избежать ошибок.

- Объект является **итератором**, если он умеет обращаться к элементам коллекции по одному за раз, при этом отслеживая своё текущее положение внутри этой последовательности.

Есть вероятность, что другой скрипт, не связанный с нашим, захочет записать свой идентификатор в наш объект buddy.

Сторонний код может создать для этого свой символ Symbol(‘id’):

```
let id = Symbol('id');
buddy[id] = "Бобик";
```

Идентификаторы, созданные с помощью Symbol, создаются уникальными всегда, вне зависимости от переданного им символьного значения названия. Но если использовать строку «id» вместо идентификатора, то это приведёт к конфликту:

```
let buddy = {name: 'Тузик'}; // Объявляем в нашем скрипте свойство 'id'
buddy.id = 'Наш идентификатор'; // ...другой скрипт тоже хочет свой
  идентификатор...
buddy.id = 'Их идентификатор' // Ой! Свойство перезаписано сторонней
  библиотекой!
```

Свойства с ключами-символами игнорируются в итераторе объектов for(... in ...) и структурой Object.keys(buddy). Это часть общего принципа сокрытия свойств с ключами-символами. Если какая-то сторонняя библиотека будет перебирать

свойства, она их также не получит. Object.assign копирует все свойства, в том числе, символьные.

```
let buddies = {
    [Symbol('Жучка')]: 'Жучка',
    [Symbol('Мурка')]: 'Мурка',
    [Symbol('Таракашка')]: 'Таракашка',
    elephant: 'Слон'
}
console.log(buddies);
let newBuddies = {};
Object.assign(newBuddies, buddies);
console.log(newBuddies);
```

```
first.js:10
{elephant: 'Слон', Symbol(Жучка): 'Жучка', Symbol(Мурка): 'Мурка', Symbol(Таракашка): 'Таракашка'} i
  elephant: "Слон"
  Symbol(Жучка): "Жучка"
  Symbol(Мурка): "Мурка"
  Symbol(Таракашка): "Таракашка"
  ▶ [[Prototype]]: Object
first.js:15
{elephant: 'Слон', Symbol(Жучка): 'Жучка', Symbol(Мурка): 'Мурка', Symbol(Таракашка): 'Таракашка'} i
  elephant: "Слон"
  Symbol(Жучка): "Жучка"
  Symbol(Мурка): "Мурка"
  Symbol(Таракашка): "Таракашка"
  ▶ [[Prototype]]: Object
```

Глобальный реестр символов

Повторные вызовы функции Symbol() приводят к созданию уникальных идентификаторов всегда, даже если мы передаём в неё одно и то же описание. А что, если нам нужно, чтобы при задаче каждому описанию соответствовал только один уникальный идентификатор? Для этого создан глобальный реестр символов. Мы создаём символы в нём, потом обращаемся к нему по этому имени. И при этом

обращении нам будет гарантированно выдаваться конкретный, соответствующий этому описанию, символ.

Для этого используется специальный метод Symbol.for(ключ). Он считывает нужный символ, а, при его отсутствии, создаст.

```
1 // читаем символ из глобального реестра и записываем его в переменную
2 let id = Symbol.for("id"); // если символа не существует, он будет создан
3 // читаем его снова и записываем в другую переменную (возможно, из другого
  места кода)
4 let idAgain = Symbol.for("id");
5 // проверяем: это один и тот же символ
6 alert( id === idAgain ); // true
```

Эти символы создаются в глобальной области видимости, поэтому доступны глобально.

Есть ещё метод Symbol.keyFor (идентификатор), возвращающий описание символа по идентификатору. Единственное, этот метод работает для глобальных символов, а при попытке поиска обычных вернёт undefined.

```
1 // получаем символ по имени
2 let sym = Symbol.for("name");
```

```
3 let sym2 = Symbol.for("id");
4 // получаем имя по символу
5 console.log( Symbol.keyFor(sym) ); // name
6 console.log( Symbol.keyFor(sym2) ); // id
```

Но мы всегда можем использовать свойство description, которое доступно для всех:

```
1 let globalSymbol = Symbol.for("name");
2 let localSymbol = Symbol("name");
3 console.log( Symbol.keyFor(globalSymbol) ); // name, глобальный символ
4 console.log( Symbol.keyFor(localSymbol) ); // undefined для неглобального
```

```
  символа

5 console.log( localSymbol.description ); // name
```

Системные символы

Имеются специальные символы, которые мы можем настраивать поведение объектов. Они содержатся внутри JavaScript и называются системными.

Вот некоторые из них:

- Symbol.hasInstance
- Symbol.isConcatSpreadable
- Symbol.iterator
- Symbol.toPrimitive

И другие. Мы позже посмотрим, как работают некоторые из них.

Итого

Тип данных Symbol — это уникальный идентификатор. Он уникален всегда, даже при использовании одного описания. Если нам нужен один идентификатор для одного описания, используем глобальные символы.

Символы скрыты для итераторов и с большой вероятностью не нарушат целостность переиспользуемого кода. Но Object.assign() скопирует объект и с символами.

Итерируемые объекты

Перебираемые или итерируемые объекты — это обобщение массивов. Концепция, позволяющая любой объект использовать в цикле for(... of ...). Естественно, массивы сами по себе являются итерируемыми. Но есть большое количество других встроенных объектов, которые являются итерируемыми. Например, строки.

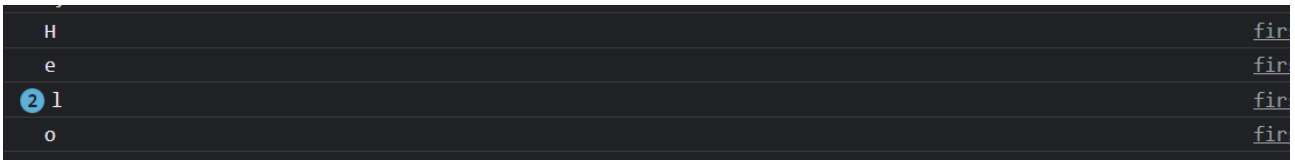
Вообще, несмотря на то что строка является примитивным типом данных в JavaScript, мы легко можем её представить как массив символов:

```
1 const string = 'Hello';
2 console.log(string[2]); // l
3 console.log(string.length); // 5
```

Этим и объясняется её простая итерируемость:

```
1 const string = 'Hello';
2 for(let str of string){
3     console.log(str);
4 }
```

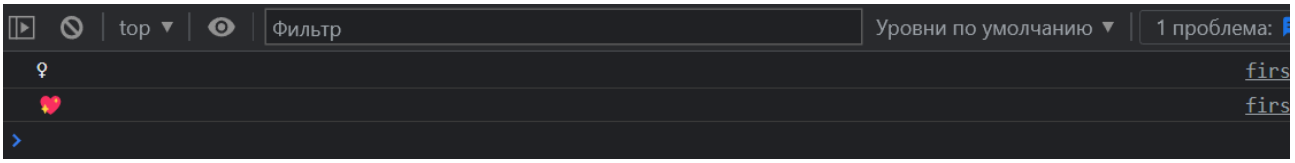
В консоли мы увидим следующие:



Это также просто работает и с другими символами, даже нестандартными, имеющими свои ASCII-коды:

```
1 const string = '♀💖';
2 for(let str of string){
3     console.log(str);
4 }
```

Консоль:



Symbol.iterator

Объект является итератором, если он умеет обращаться к элементам коллекции по одному за раз, при этом отслеживая своё текущее положение внутри этой последовательности.

В JavaScript итератор — это объект, который предоставляет метод next(), возвращающий следующий элемент последовательности. Этот метод возвращает объект с двумя свойствами: done и value.

Если объект не является массивом, но является коллекцией каких-то элементов, то он вполне подходит для цикла for(... of ...).

Например, у нас есть объект с диапазоном чисел:

```
let range = {
    from: 1,
    to: 17
};
```

Мы хотим из него сделать последовательность с таким образом: for(let number of range), где на выходе получим последовательность от 1 до 17. Чтобы это сделать, нам нужно добавить в последовательность объект с именем Symbol.iterator.

Приведём пример и посмотрим, что происходит:

```
let range = {
    from: 1,
    to: 17
};
// 1. вызов for..of сначала вызывает эту функцию
range[Symbol.iterator] = function() {
    // ...она возвращает объект итератора:
// 2. Далее, for(..of..) работает только с этим итератором, запрашивая у
  него новые значения
    return {
        current: this.from,
        last: this.to,
// 3. next() вызывается на каждой итерации цикла for(..of..)
        next() {
// 4. он должен вернуть значение в виде объекта {done:.., value :...}
```

```
15         return this.current <= this.last ?  { done: false, value:
   this.current++ } : { done: true };
16         }
17     };
18 };
19 // теперь работает!
20 for (let number of range) {
21     console.log(number);
22 }
```

Когда цикл for(... of ...) запускается, он вызывает Symbol.iterator один раз (или выдаёт ошибку, если метод не найден). Этот метод должен вернуть итератор — объект с методом next. Далее цикл работает только с возвращённым объектом от Symbol.iterator. Для получения следующего значения, цикл запускает метод next(), который должен возвратить данные в формате объекта {done: boolean, value: any}. Формат объекта показан в типах интерфейсов Typescript, так понятнее, что именно мы должны увидеть. Тип Any означает, что тип у нас может быть в виде любого из восьми типов JavaScript, в том числе, и в объекте.

Обратим внимание на ключевую особенность итераторов — это разделение ответственности. У объекта range нет метода next, этот метод есть у другого объекта — итератора, именно его next и генерирует значения. В итоге, сам объект итератор отделён от итерируемого объекта. Это позволяет снимать ограничения, связанные с одновременным выполнением нескольких итераторов, даже асинхронных. Попробуем объединить итератор и итерируемый объект и посмотрим, что получится:

```
1 let range = {
2     from: 1,
3     to: 17,
4     [Symbol.iterator]() {
5         this.current = this.from;
6         return this;
7     },
```

```
 8     next() {
 9         return this.current <= this.to ?  { done: false, value:
   this.current++ } : { done: true };
10     }
11 };
12 for (let number of range) {
13     console.log(number);
14 }
```

Получилось короче? Несомненно. Но при использовании двух таких итераторов одновременно, у них будет один контекст выполнения this, результат выполнения будет ошибкой расчётов. Конечно, необходимость одновременного запуска нескольких циклов for(... of ...) возникает редко, но разделение ответственности помогает нам писать код без заложенных трудноуловимых багов.

Array.from

Введём определение псевдомассива. Псевдомассивом назовём объект, который имеет индексы и свойство length. Как мы видели выше, строки можно представить в виде псевдомассива, хотя они не являются объектами. А вот такой объект является псевдомассивом, но, в отличие от строк, итерировать его нельзя:

```
1 let pseudo={
2     0: 'first',
3     1: 'second',
```

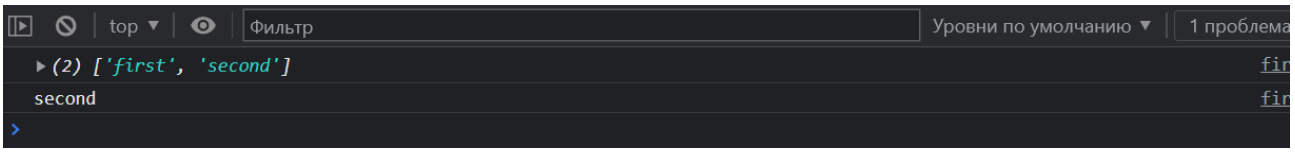
```
4     length: 2
5 }
```

Сделать массив из подобного псевдомассива или итерируемого объекта поможет метод Array.from.

```
1 let pseudo={
2     0: 'first',
3     1: 'second',
```

```
4     length: 2
5 }
6 let array = Array.from(pseudo);
7 console.log(array);
8 console.log(array.pop());
```

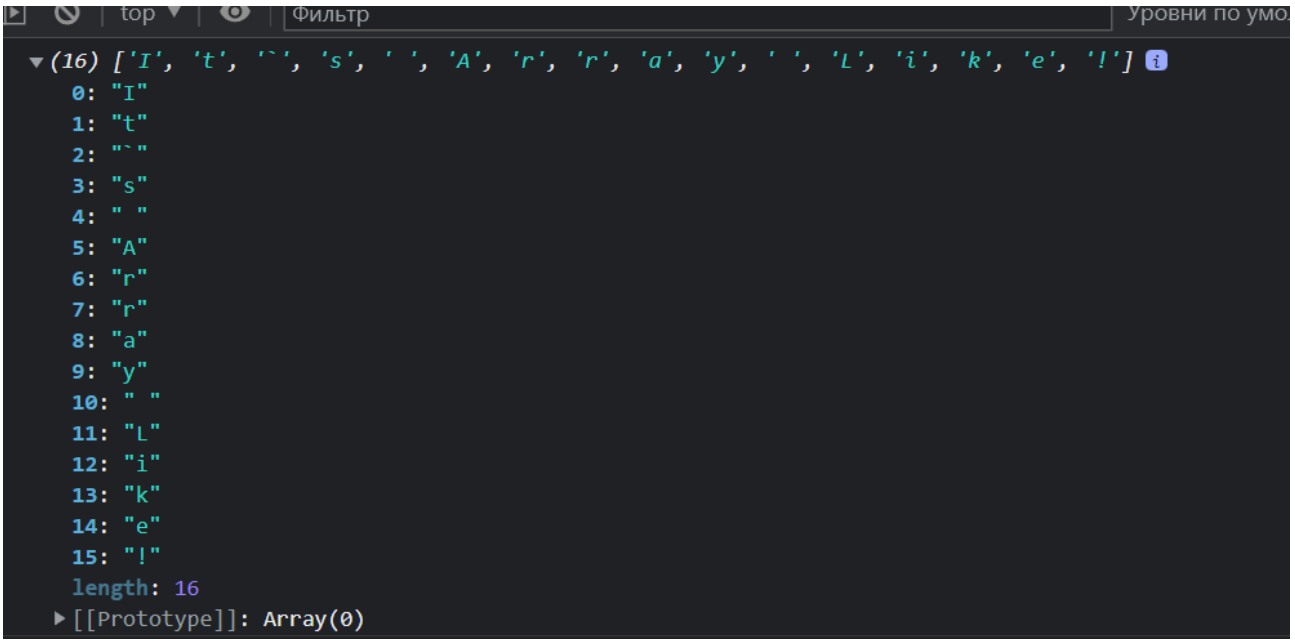
Что выведется в консоли:



И то же самое со строкой:

```
1 let pseudo = 'It`s Array Like!';
2 let array = Array.from(pseudo);
3 console.log(array);
```

Консоль:



Map и Set. WeakMap и WeakSet

Для более удобного решения повседневных задач нам иногда может понадобиться весь функционал коллекций. Рассмотрим их подробнее.

Map

Map — это коллекция ключ/значение, как и Object. Но основное отличие в том, что Map позволяет использовать ключи любого типа.

Методы и свойства:

- new Map() — создаёт коллекцию
- map.set(key, value) — записывает по ключу key значение value
- map.get(key) — возвращает значение по ключу или undefined, если ключ key отсутствует
- map.has(key) — возвращает true, если ключ key присутствует в коллекции, иначе false
- map.delete(key) — удаляет элемент (пару «ключ/значение») по ключу key
- map.clear() — очищает коллекцию от всех элементов
- map.size — возвращает текущее количество элементов

Например:

```
let map = new Map();
map.set("1", "str1");    // строка в качестве ключа
map.set(1, "num1");      // цифра как ключ
map.set(true, "bool1");  // булево значение как ключ
// помните, обычный объект Object приводит ключи к строкам?
// Map сохраняет тип ключей, так что в этом случае сохранится 2 разных значения:
console.log(map.get(1)); // "num1"
console.log(map.get("1")); // "str1"
```

```
 9
10 console.log(map.size); // 3
```

Мы видим, что ключи у нас могут принимать любой тип. Поэтому, с точки зрения использования коллекций, неправильно обращаться к ним через map[ключ]. Хотя это будет работать. Правильно обращаться через геттер и сеттер — get и set.

Map, в отличие от обычных объектов, может использовать объекты вместо ключей.

Map.set возвращает при вызове объект Map, поэтому мы для установки значений можем использовать цепочку из методов set:

```
1 let map = new Map();
2 map.set("1", "We")
3   .set(1, "likes")
4   .set(true, "JS");
5 console.log(map);
```

Вот что будет в консоли:

```
▼Map(3) {'1' => 'We', 1 => 'likes', true => 'JS'}
  ▼[[Entries]]
    ▶0: {"1" => "We"}
    ▶1: {1 => "likes"}
    ▶2: {true => "JS"}
     size: 3
  ▶[[Prototype]]: Map
```

Перебор коллекции Map

В коллекцию map перебор происходит в том же порядке, в каком добавлялись элементы

Для перебора коллекции Map есть 4 метода:

- map.keys() — возвращает итерируемый объект по ключам
- map.values() — возвращает итерируемый объект по значениям
- map.entries() — возвращает итерируемый объект по парам вида [ключ, значение], этот вариант используется по умолчанию в for(..of..)
- map.forEach() — итератор, работающий так же, как и с массивом

Например:

```
let recipeMap = new Map([
  ["огурец", 500],
  ["помидор", 350],
  ["лук",    50]
]);
// перебор по ключам (овощи)
for (let vegetable of recipeMap.keys()) {
    console.log(vegetable); // огурец, помидор, лук
}
// перебор по значениям (числа)
for (let amount of recipeMap.values()) {
    console.log(amount); // 500, 350, 50
}
// перебор по элементам в формате [ключ, значение]
for (let entry of recipeMap) { // то же самое, что и recipeMap.entries()
    console.log(entry); // огурец,500 (и так далее)
}
```

```
// выполняем функцию для каждой пары (ключ, значение)
recipeMap.forEach((value, key, map) => {
    console.log(`${key}: ${value}`); // огурец: 500 и так далее
  }
);
```

Работа с объектами

Мы можем создавать коллекции Map из объектов и, наоборот, объекты из Map.

Object.entries поможет создать Map:

```
let map = new Map(Object.entries(obj));
```

Object.fromEntries поможет создать объект из Map:

```
let obj = Object.fromEntries(map);
```

Set

Объект Set — это особый вид коллекции: «множество» значений (без ключей), где каждое значение может появляться только один раз.

Его основные методы это:

- new Set(iterable) — создаёт Set, и если в качестве аргумента был предоставлен итерируемый объект (обычно это массив), то копирует его значения в новый Set
- set.add(value) — добавляет значение (если оно уже есть, то ничего не делает), возвращает тот же объект set
- set.delete(value) — удаляет значение, возвращает true, если value было во множестве на момент вызова, иначе false
- set.has(value) — возвращает true, если значение присутствует во множестве, иначе false
- set.clear() — удаляет все имеющиеся значения
- set.size — возвращает количество элементов во множестве

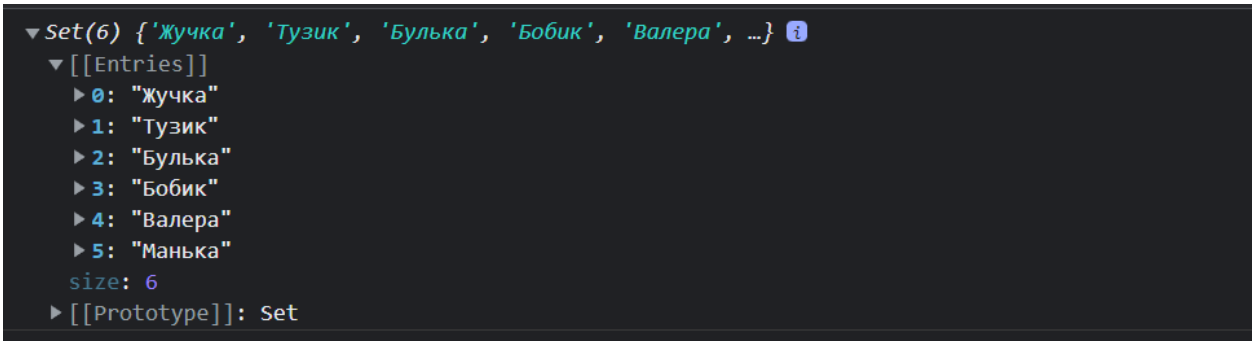
Основная «изюминка» — это то, что при повторных вызовах set.add() с одним и тем же значением ничего не происходит, за счёт этого как раз и получается, что каждое значение появляется один раз. Создание Set из массива с повторяющимися элементами также удалит повторные элементы, сделав коллекцию из уникальных элементов. Это самое частое применение коллекции Set.

Проверим это на практике. Предположим, мы смотрим, какие собачки подбегали к нам взять косточку. Но некоторые собачки хитрые и подбегали несколько раз. Узнаем, кто точно к нам подбегал:

```
let buddies = [
    'Жучка',
    'Тузик',
```

```
    'Булька',
    'Тузик',
    'Бобик',
    'Жучка',
    'Валера',
    'Жучка',
    'Тузик',
    'Манька'
];
let uniqueBuddies = new Set(buddies);
console.log(uniqueBuddies);
```

Что нам выведется в консоль:



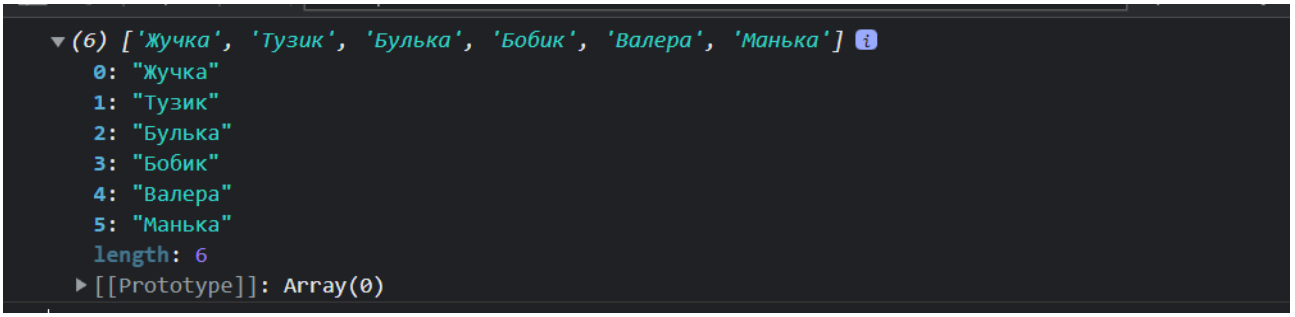
Мы увидим, что у нас взяли 10 косточек, а собак было всего 6.

Перевести обратно в массив нам поможет уже известный нам метод Array.from:

```
let buddies = [
    'Жучка',
    'Тузик',
    'Булька',
    'Тузик',
    'Бобик',
    'Жучка',
```

```
    'Валера',
    'Жучка',
    'Тузик',
    'Манька'
];
let uniqueBuddies = new Set(buddies);
let arr = Array.from(uniqueBuddies);
console.log(arr);
```

В итоге, у нас будет массив из уникальных кличек собак:



Перебор элементов объекта Set происходит схоже с перебором у Map специально для совместимости между ними. И теми же методами: forEach, keys, values, entries.

WeakMap и WeakSet

Кратко рассмотрим «побратимов» Map и Set. Необходимость в их использовании возникает очень редко, поэтому подробнее рассматривать нет смысла. Подробнее об их использовании можно посмотреть в документации: WeakMap и WeakSet.

WeakMap — это Map-подобная коллекция, позволяющая использовать в качестве ключей только объекты, и автоматически удаляющая их вместе с соответствующими значениями, как только они становятся недостижимыми иными путями.

WeakSet — это Set-подобная коллекция, которая хранит только объекты и удаляет их, как только они становятся недостижимыми иными путями.

Удаляются объекты автоматически сборщиком мусора. Рассматриваемые коллекции не создают специальных ссылок на эти объекты и не препятствуют их удалению.

Обе этих структуры данных не поддерживают методы и свойства, работающие со всем содержимым сразу или возвращающие информацию о размере коллекции. Возможны только операции на отдельном элементе коллекции.

WeakMap и WeakSet используются как вспомогательные структуры данных в дополнение к «основному» месту хранения объекта. Если объект удаляется из основного хранилища и нигде не используется, кроме как в качестве ключа в WeakMap или в WeakSet, то он будет удалён автоматически.

Модули

Ранее добавить ещё один JS-файл можно было только через html-тег script:

```
<script src="first.js"></script>
```

Мы добавляли библиотеки, например, jQuery для облегчённого написания уже имеющихся в языке возможностей. Но JavaScript не стоит на месте, и концепция подключения модулей продолжила своё развитие.

Скрипты росли, делались сложнее и со временем стало труднее в них ориентироваться. Стало неудобно добавлять код в глобальную область, как делает тег script, поэтому появились сборщики скриптов типа Webpack, затем появились и полноценные среды со своей экосистемой. Они и научились первыми собирать модули JavaScript, а с новыми стандартами ECMAScript, этому научился и сам язык.

Экспорт

Чтобы получить доступ к объектам модулей, надо их экспортировать. В подключаемых модулях мы можем пометить, какие именно элементы мы хотим сделать доступными для экспорта. Для этого надо перед ними поместить ключевое слово export:

```
export const name = 'square';

export function draw(ctx, length, x, y, color) {
  ctx.fillStyle = color;
  ctx.fillRect(x, y, length, length);
```

```
6
7   return {
8     length: length,
9     x: x,
10    y: y,
11    color: color
12  };
13 }
```

Экспортировать объекты мы можем только на верхнем уровне видимости, внутри блоков этого делать нельзя. Экспортировать мы можем переменные, функции и классы.

Мы можем указать все экспортируемые объекты в начале модуля для удобства:

```
1 export { name, draw, reportArea, reportPerimeter };
```

Импорт

После того как мы пометили в подключаемых модулях объекты, доступные для экспорта, нам нужно как-то подключить их там, где мы хотим их использовать. Это мы можем сделать с помощью конструкции import ... from '...':

```
1 import { name, draw, reportArea, reportPerimeter } from './modules/square.js';
```

После ключевого слова import мы в деструктуризованном объекте перечисляем имена импортируемых объектов. Если мы хотим импортировать всё, что можно, то можно просто указать звёздочку. Но для неё лучше указать имя, к которому мы будем вызывать для обращения к этому объекту с помощью оператора as:

```
1 import * as Square from './modules/square.js';
```

После ключевого слова from мы указываем относительный путь до файла, содержащего импортируемый модуль.

Точка в начале пути мы можем использовать для обозначения текущей директории. Две точки — директория выше. Тем самым, мы сокращаем длину написанного пути.

Например:

```
/js-examples/modules/basic-modules/modules/square.js
```

Становится:

```
./modules/square.js
```

Далее нам необходимо подключить модуль main.js на нашу HTML-страницу. Это очень похоже на то, как мы подключаем обычный скрипт на страницу, с некоторыми заметными отличиями.

Прежде всего, вам нужно добавить type="module" в <script>-элемент, чтобы объявить, что скрипт является модулем. Чтобы подключить модуль main.js, нужно написать следующее:

```
<script type="module" src="main.js"></script>
```

Мы можем также встроить скрипт модуля непосредственно в HTML-файл, поместив JavaScript-код внутрь <script>-элемента:

```
<script type="module">
    /* код JavaScript модуля */
</script>
```

Скрипт, в который мы импортируем функционал модуля, в основном действует как модуль верхнего уровня.

Мы можем использовать инструкции import и export только внутри модулей, внутри обычных скриптов они работать не будут. Подключаемые объекты модулей доступны только для чтения.

Отличия модулей от обычных скриптов

1. Подключаемые модули — это функционал, который был добавлен недавно. Он работает автоматически в строгом режиме — режиме, который использует последние нововведения языка. Из-за этого какой-то код может сработать неожиданно.
2. Могут возникнуть ошибки CORS при локальной загрузке файлов HTML. Нужно проводить тестирование этих файлов через сервер.

3. В теге script при загрузке модуля по умолчанию устанавливается атрибут defer.
4. Модули подключаются только один раз, даже если есть несколько ссылок в тегах script для их подключения
5. Объекты из модуля добавляются в область видимости одного скрипта. В глобальной области они не видны.

Некоторые дополнительные возможности

- Экспорт по умолчанию. Некоторые модули объявляют экспортные объекты с помощью ключевого слова default. Импорты делаются без фигурных скобок — так разработчики решили упростить использование экспортируемого функционала сторонним модулем, а также помогает модулям JavaScript взаимодействовать с существующими модульными системами CommonJS и AMD. Подробнее можно почитать здесь.
- Для импорта всех экспортируемых объектов из файла вместо перечисления имён объектов можно указать звёздочку *.
- Для избежания конфликтов имён в текущем файле можно переименовывать экспортируемые объекты с помощью ключевого слова as:

```
1 import { name as title, draw as picture } from './modules/square.js';
```

- Самая свежая возможность модулей — динамическая загрузка. Выполняется она с помощью промиса import(). Это даёт нам возможность воспользоваться преимуществами ленивой загрузки модулей, когда они загружаются именно тогда, когда они необходимы. Это снижает нагрузку на браузер и облегчает выполнение скриптов.

Подведём итоги

В наших программах, так или иначе, работают фоновые процессы. Мы никак не можем повлиять на их работу, их действие неотвратимо. Один из них — сборщик мусора. Он следит за высвобождением неиспользуемой памяти, чтобы нашему скрипту и другим программам на компьютере она была доступна.

Работает он по принципу доступности ссылок на объект. Есть базовые, корневые ссылки, от них ссылки идут на другие объекты, он их обходит по принципу обхода деревьев. И те объекты, до которых он не доберётся, будут удалены.

Далее мы рассмотрели подробнее новый тип данных Symbol и особенности работы с ним. Узнали, что данные с идентификатором Symbol вместо ключа будут скрыты для итераторов, что позволяет защитить данные от перезаписи сторонними скриптами.

Разобрали работу итераторов в JavaScript и создание своих итераторов для объектов. Попробовали сделать массивы из объектов, похожих на массив.

Поработали с коллекциями и разобрали их дополнительные возможности. Узнали, что в коллекциях Map можно записывать в качестве ключей объекты, а коллекция Set формируется только из уникальных элементов. Разобрали методы для работы с коллекциями.

В конце урока мы рассмотрели базовые возможности JavaScript по работе с модулями — дополнительными файлами с кодом, библиотеками. Подробнее все возможности модулей вы разберёте уже в курсах фреймворков, синтаксис их использования несколько отличается в каждом из них. Здесь мне хотелось бы донести то, что программирование не стоит на месте, фронтэнд же развивается семимильными шагами. Появляются новые стандарты, развиваются фреймворки. Раньше в JavaScript работа с файлами была сильно ограничена настройками безопасности. Сейчас, с развитием технологий, появилась возможность достаточно широко работать с файлами. Более того, появился язык Node.js, позволяющий писать бэкенд на языке JavaScript.